Namaste JavaScript – Season 1

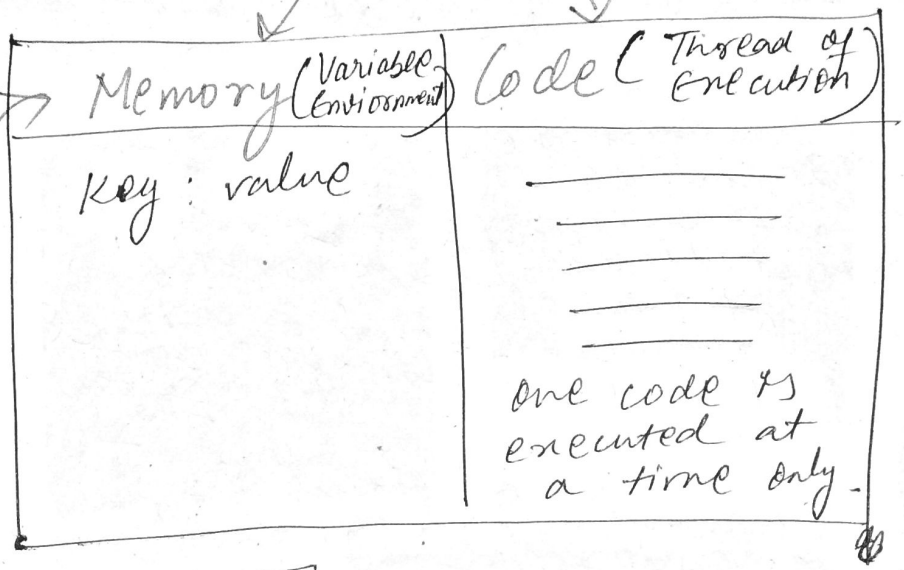
04/01/24

1. Execution Context.
2. How JS is executed & Call Stack.
3. Hoisting in JS (variables & Functions)
4. Functions and Variable Enviornment.
5. Shortest JS Program, window & this Keyword.
6. undefined & not defined in JS
7. The Scope Chain, Scope & Lexical Enviornment.
8. let & const in JS, Temporal Dead Zone.
9. Block Scope & Shadowing in JS.
10. Closure in JS
11. setTimeout + Closure Interview Questions
12. Famous Interview Questions ft. Closures
13. First Child Function f.t Anonymous Function
14. Callback Function in JS f.t Event Listeners
15. Asynchronous JavaScript & Event Loop from Scratch.
16. JS engine exposed, Google's V8 architecture.
17. Trust Issues with setTimeout()
18. Higher Order Function f.t Functional Programming
19. map, filter and reduce.

1. How JS works & Execution Context

Everything in JS happens in the Execution Context.

Execution Context.



all variables & Functions are stored in key-value pair

"JS is a synchronous single-threaded language"

single-threaded means JS can execute one code at a time.

&, Synchronous single-threaded language means JS can execute one code at a time in a specific order.

Type of Execution Context.

Global
- It is created when JS starts to run

Function
- It is created when function is called

2. How JS Code is Executed? Call Stack.

What happens when you run a JS code?
→ An Execution Content is created.

Take an example:
Suppose you run this code,

```
var n = 2;
function square(num) {
  var ans = num * num;
  return ans;
}
var square2 = square(n);
var square4 = square(4);
```

Execution Content is Created using 2 phases —
→ Memory Creation Phase.
→ Code Execution Phase.

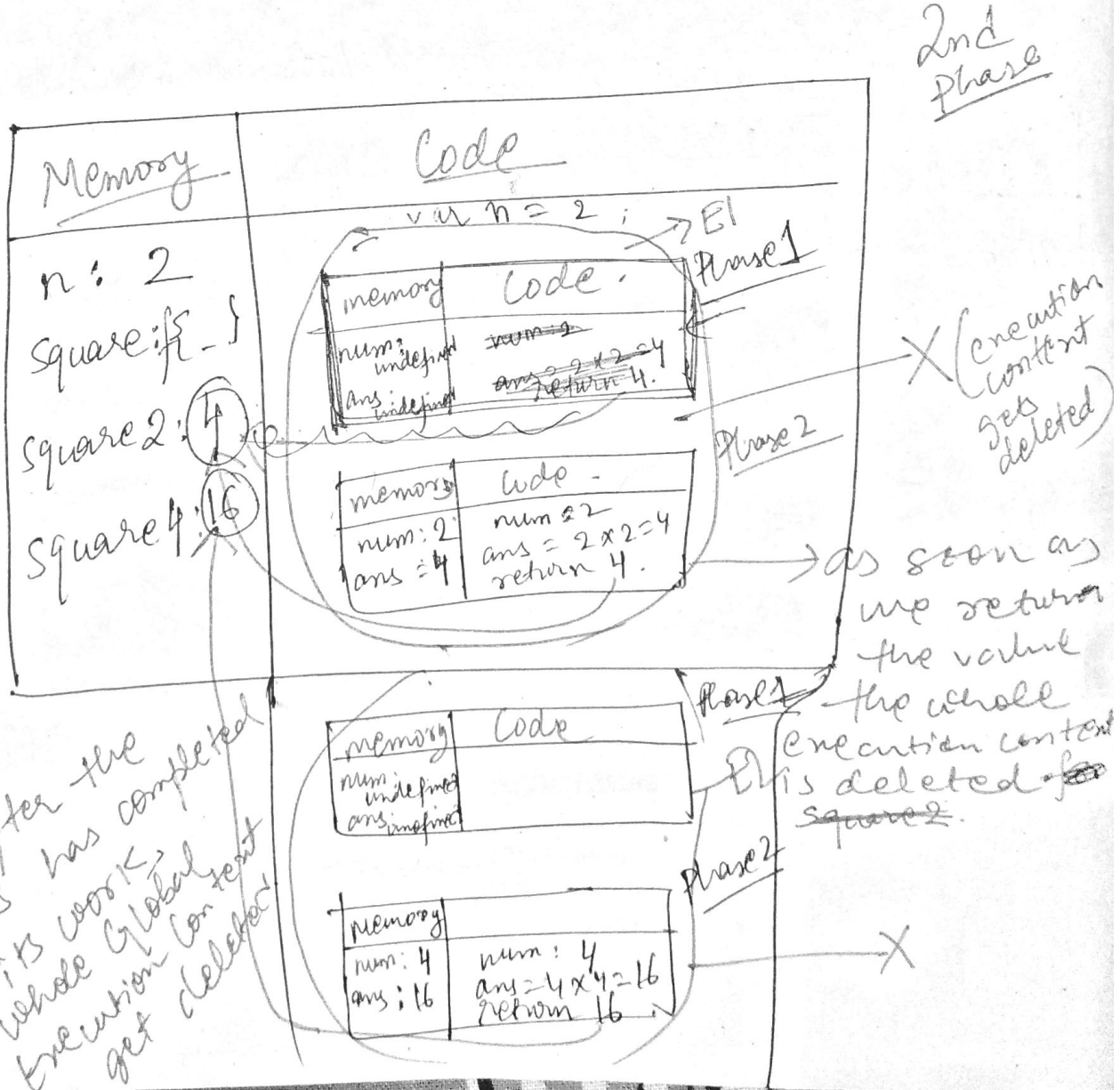
Global Execution Context (1st Phase).

Memory	Code
n : undefined square : {f .....} (whole code of function is stored) square2 : undefined square4 : undefined	

2nd Phase - (code execution)

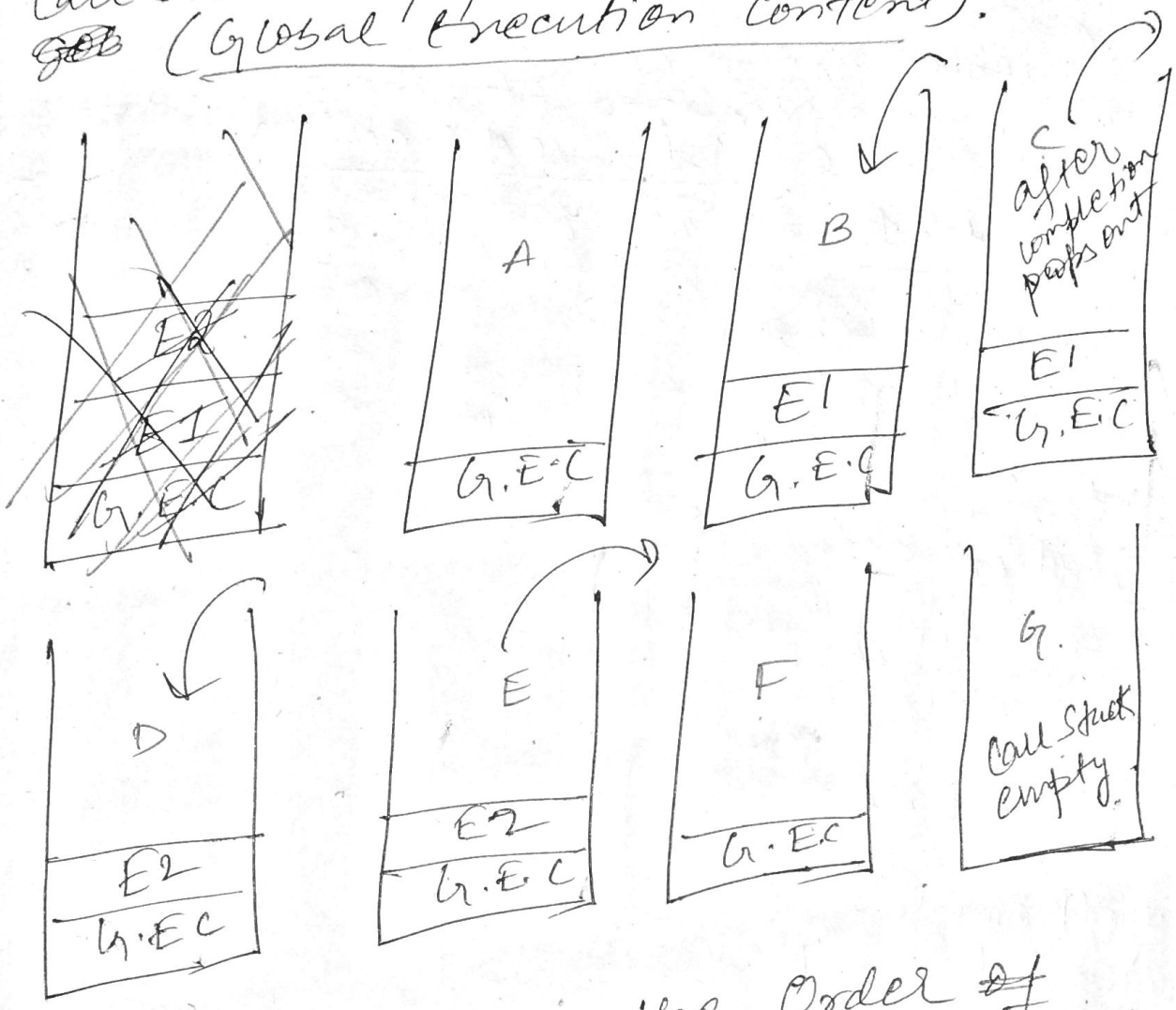
Now in 2nd phase, it starts ~~going~~ through the whole code line by line. As it encounters var n = 2, it assigns 2 to 'n'. Until now the value of 'n' was undefined. For function, there is nothing to execute..

Coming to line 6 i.e, var square2 = square(n); here functions are a bit different than any other language. A new execution context is created altogether.



Call Stack - JS manages the execution context with the help of a call stack.

Whenever a JS code is run the call stack is populated with the G.E.C (Global Execution Context).



→ CallStack maintains the Order of execution of execution contexts.

→ CallStack is also known as Execution Context Stack / Program Stack / Control Stack / Runtime Stack / Machine Stack.

3. Hoisting in JS (variables & functions).

Hoisting is a phenomenon in JS by which we can access the variables and functions even before you have initialized it, and this happen due to 1st phase of execution context.
↳ memory creation phase.

Example : 1

```
getName();  // op -> Namaste JS
console.log(x);  // op -> undefined
console.log(getName);  // f getName(){.....}
var x = 7;
function getName(){
  console.log("Namaste JS");
}
```

If this wasn't present then the console.log(x) would have thrown error i.e not defined

Example : 2

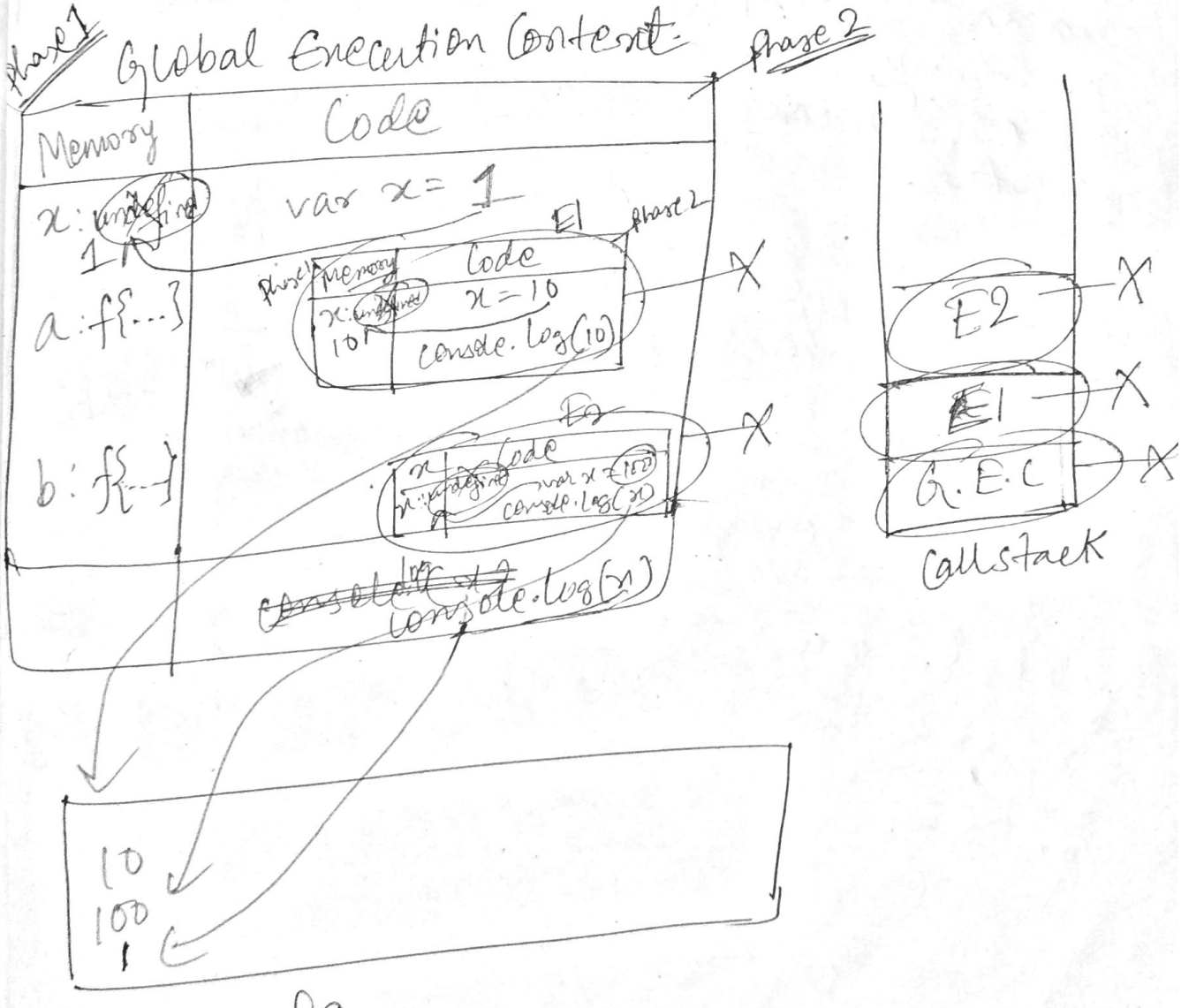
```
getName();  // Error, getName() is not a function it is a variable
console.log(getName);
var getName = function(){
  console.log("Namaste JS");
};
```

4. Functions & Variable Environment

```
var x = 1
a();
b();
console.log(x);
function a(){
  var x = 10;
  console.log(x);}
function b(){
  var x = 100;
  console.log(x);}
```

Output:

10
100
1



5. Shortest JS Program

↳ Empty JS file

* ~~The Shortest~~ As also in this case JS creates the GEC which has memory space and the execution context.

* JS engine creates something known as 'window'. It is an object which is created in Global Space. It contains lots of functions and variables and can be accessed anywhere from the program. JS engine also creates a this keyword, which points to the window object at global level. So, in summary, along with GEC, a global object (window) and a this variable are created.

* If we create any variables in the global scope, then the variables get attached to the global object.

* At global level, `this === window`

window (Browser)

global space/scope → Anything which is not inside the function is in the global space

6. Undefined vs Not-defined in JS.

(*) In first phase (memory allocation) JS assigns each variable a placeholder called undefined.

(*) undefined is when memory is allocated for the variable, but no value is assigned yet.

(*) If an object/variable is not even declared/found in memory allocation phase, and tried to access it then it is not defined.

E.g

```
console.log(x); // undefined
var x = 25;
console.log(x); // 25
console.log(a); // a is not defined
```

7. The Scope Chain, Scope & Lexical Environment :- (V. Imp)

→ Scope in JavaScript is directly related to Lexical Environment.

```
// Case-1
function a() {
    console.log(b); // 10
// Instead of printing undefined it prints
10, So somehow this a function could
access the variable b outside the function
scope.
}
var b = 10;
a();
```

// Case-2:

```
function a(){
    c();
    function c(){
        console.log(b); //10
    }
}
var b = 10;
a();
```

10 is printed. It means that within nested function too, the global scope variable can be accessed

~~→ 10 is printed~~

// Case-3:

```
function a(){
    c();
    function c(){
        var b = 100;
        console.log(b); // 100
    }
}
var b = 10;
a();
```

100 is printed meaning local variable of the same name took precedence over a global variable.

// Case-4

```
function a(){
    var b = 10;
    c();
    function c(){
        console.log(b); //10
    }
}
```

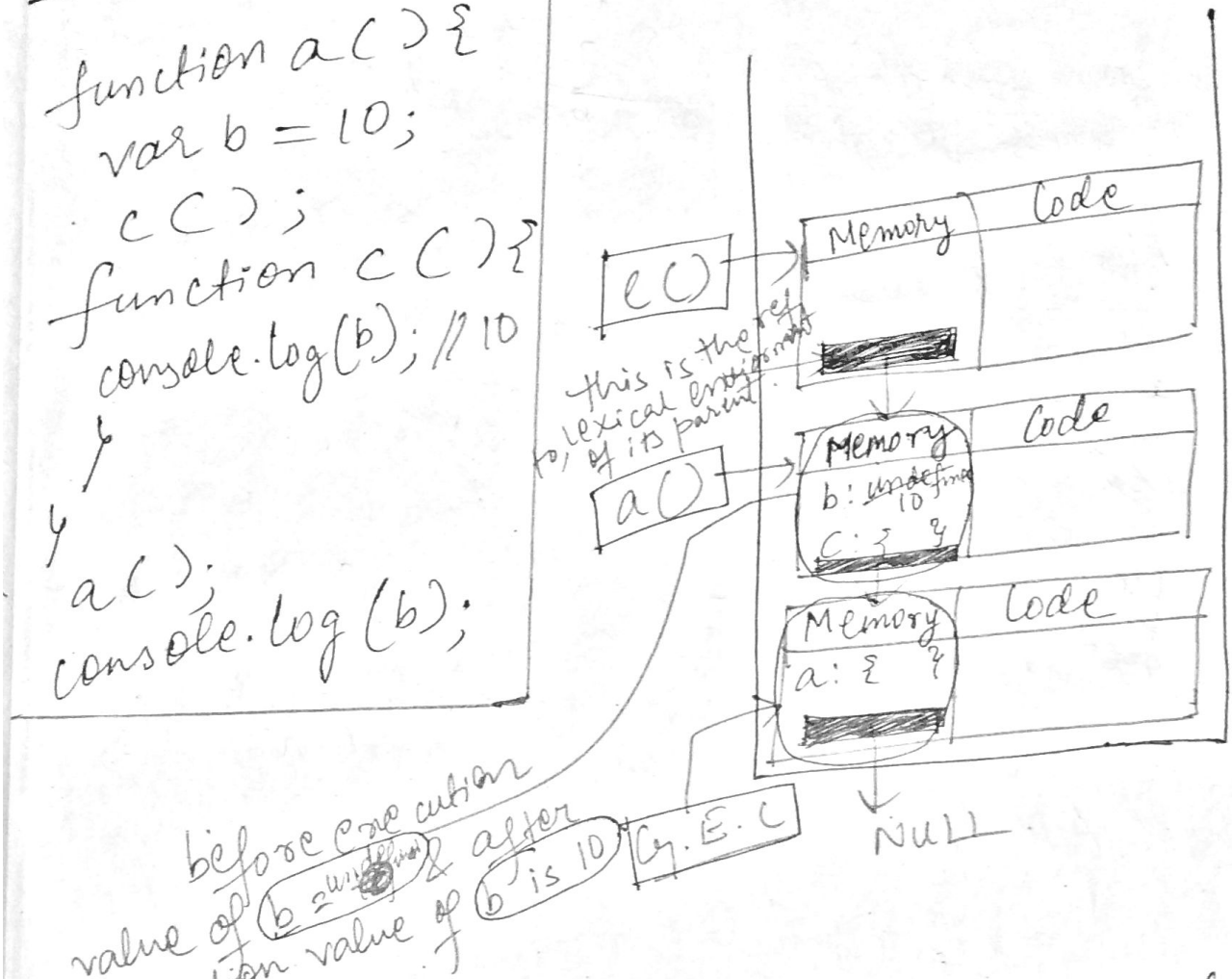
A function can access a global variable, but the global execution context can't access any local variable.

```
a();
console.log(b); // Error, Not Defined
```

Scope means where you can access a specific variable or function in a code.

```
function a(){
  var b = 10;
  c();
  function c(){
    console.log(b); // 10
  }
}
a();
console.log(b);
```

Call Stack



before execution value of b = undefined & after execution value of b is 10

Note :- (i) whenever a execution context is created a Lexical Enviornment is also created.

(ii) Lexical Enviornment is the local memory along with the lexical enviorment of its parent.

What is the meaning of 'Lexical'?

→ 'Lexical' as a term means 'in a order' or 'in a sequence'.

If we say in terms of code then, function c() {} → is present Lexically inside function a() and function a() {} → is lexically inside the global scope.

Scope Chain / Lexical Environment Chain

The process of going one by one to parent and checking for values is called scope chain or Lexical environment chain.

8. 08/04/24

Let & Const in JS, Temporal Dead Zone

→ What is a temporal dead zone?

→ Are let & const declaration hoisted?

→ Difference btw Syntax Error vs Reference Error vs Type Error?

* Let & const declarations are hoisted in JS, but they are different from var.

Example:

```
console.log(a); // Reference error: Cannot access 'a' before initialization.
console.log(b); // Prints undefined
let a = 10;
console.log(a); // 10
var b = 15;
console.log(window.a); // undefined
console.log(window.b); // 15
```

It looks like let isn't hoisted, but it is, let's understand,

→ Both a and b are actually initialized as undefined in hoisting stage. But variable b is inside the storage space of GLOBAL, and a is in a separate memory object called script, where it can be accessed only after assigning some value to it first, i.e one can access 'a' only if it is assigned. Thus it throws error.

Temporal Dead Zone:

↳ Time since ~~when~~ the "let" variable was hoisted until it is initialized some value.

So, any line before "let a = 10" is in the Temporal Dead Zone for 'a'.

↳ Reference Error are thrown when variables are in the Temporal Dead Zone.

Reference Error: If, I try to access any random variable, which is not present in the program, it throws reference error.

e.g.

```
console.log(x) // Reference error: not defined
var a = 100;
let b = 10
```

'let' is a strict version of 'var'.

E.g.

```
let a = 10;
let a = 100; // this code is rejected as Syntax Error.
             (Duplicate Declarations)
```

```
let a = 10;
var a = 100; // this code also rejected as Syntax Error
             (Can't use same name in same scope).
```

'const' is even more stricter than 'let'.

E.g.

```
let a;
a = 10;
console.log(a); // 10. Note declaration and assigning of a is in different lines
```

```
const b;
b = 10;
console.log(b); // Syntax error: Missing initializer in const declaration
```

→ This type of declaration doesn't work, `const b = 10` → will only work. (It should be declared & initialized together only).

```
const b = 10;
b = 100;
console.log(b);
```

→ TypeError: Assignment to constant variable ~~(It should be declared & initialize~~

(Once, const variable is declared & initialized it cannot ~~be~~ re-assigned new value).

Some Good Practices

- Try using const wherever possible.
- If, not, use let, avoid var.
- Declare and initialize all variables with let to the top to avoid errors to shrink temporal dead zone window to zero.

9. Block Scope & Shadowing in JS -

What is a Block in JavaScript?

→ ✱ Code inside curly bracket is called block. { }

✱ Multiple statements are grouped inside a block so it can be written where JS expects single statements like in if-else, loops, functions, objects etc.

```
if (true) {
  let a = 100
  console.log(a);
}
```

Block

What is a Block Scope and Lexical Scope Chain?

→ Block Scope means what all variables and functions we can access inside this block.

```
{
  var a = 10;
  let b = 20;
  const c = 30;
}
```

} Hoisted in Global scope.

} Here, let and const are hoisted in the Block Scope.

let & const cannot be accessed outside the block scope, but var can be accessed outside the block.

Shadowing in JavaScript:

```
var a = 100;
{
  var a = 10;
  let b = 20;
  const c = 30;
  console.log(a); // 10
  console.log(b); // 20
  console.log(c); // 30
}
console.log(a); // 10, instead of 100 we got 10. Since variable 'a' is defined with var keyword and initially in global space a = 100, but when a = 10 assigned inside the Block, then it replaces 100 with 10 in global space itself.
```

The variable inside the block shadows the outside variable. This happens only for var.

Illegal Shadowing:

```
let a = 20;
{
  var a = 20;
}
// Syntax Error: Identifier 'a' is already been declared.
```

- We cannot shadow let with var. But it is valid to shadow a let using a let. However, we can shadow var with let.
- All scope ~~rules~~ rules that work in function are same in arrow function too.

- Lexical Scope also works in the same way inside the block also.
- All scope rules that work with functions are exactly same with Arrow functions also.

10. Closures in JS.

- Function ~~bbbb~~ bundled along with it's lexical scope is closure.
- JS has a lexical scope environment. If a function needs to access a variable, it first goes to its local memory. When it does not find it there, it goes to the memory of its lexical parent.

E.g:

```
function x (){
    var a = 7;
    function y (){
        console.log(a);
    }
    return y;
}
var z = x();
console.log(z); // value of z is entire code of function y.
```

Over here function y along with its lexical scope i.e (function x) would be a closure.

when y is returned, not only is the function returned but the entire closure is returned and put inside z.

Another Example:

```
function z() {
    var b = 900;
    function x() {
        var a = 7;
        function y() {
            console.log(a, b); }
        y();
    }
    x();
}
z(); // o/p -> 7 900
```

In, Simple words:

A, closure is a function that has access to its outer function scope, even after the function has returned.

Meaning -> A closure can remember and access variables and arguments reference of its outer function even after the function has returned.

09/01/24 11. SetTimeout + Closures Interview Questions.

remember: Time - Tide & Javascript waits for none.

```
function x() {
    var i = 1;
    setTimeout(function() {
        console.log(i);
    }, 3000);
    console.log("Namaste JS");
}
x();
// Output:
Namaste JS
1
```

→ we expect JS to wait 3 sec, print 1 and then print, "Namaste JS". But JS, prints the string immediately, waits 3 sec and then prints 1.

→ This is because the function inside setTimeout forms a closure. So, wherever function goes it carries this ref along with it.

→ setTimeout takes this callback function & attaches timer of 3000ms and stores it. Goes to the next line and print the string without waiting.

→ After 3000ms runs out, JS takes function, put it into call stack and runs it.

Q. Print 1 after 1 sec, 2 after 2 sec, 3 after 3 sec, 4 after 4 sec ... till 5 after 5 sec. (Interview Question)

→ **Wrong Way**

```
function x(){
for(var i=1; i<=5; i++){
setTimeout(function(){
    console.log(i);
}, i*1000); }
console.log("Namaste JS");
}
x();
```

Output:

Namaste JS

6
6
6
6
6

— Wrong Ans

This happens because of closures. When setTimeout stores the function somewhere and attaches timer to it, the function remembers its reference to 'i', (*)

(*) not value of 'i'. All 5 copies of function points to the same ref of 'i'. JS stores these 5 functions, prints the string and then comeback to the functions. By then the timer has run fully. And due to looping, the i value became 6. And when the callback function runs, the variable i=6. So, 6 is printed at each console.log.

To avoid this, we can use let instead of var as let has Block Scope. For each iteration, the 'i' is a new variable altogether (new copy). Everytime setTimeout is run the inside function forms closure with new variable 'i'.

Correct way:

```
function x(){
for(let i=1; i<=5; i++){
setTimeout(function(){
  console.log(i);
}, i*1000);
}
console.log("Namaste JS");
}
x();
```

Output:

Namaste JS
1
2
3
4
5

// Tricky Part → Implement using var only.

```
function x(){
for(var i=1; i<=5; i++){
  function close(x){
    setTimeout(function(){
      console.log(x);
    }, x*1000);
  }
  close(i); // everytime you call close() it creates new copy of i. Only this time, it is with var itself.
}
}
x();
```

12. Famous Interview Questions, ft. Closures

Q1) What is closure in JS?] done.
Q2) Can you give an example.]
Q3) Use of double parenthesis ()() in JS.

```
function outer() {
    var a = 10;
    function inner() {
        console.log(a);
    } // inner forms a closure with outer
    return inner;
}
outer()();
```

outer()(); → 2nd parenthesis calls the inner function.

↓

1st parenthesis call return inner function

Q4) Will the below code still forms a closure?

→

```
function outer(){
    function inner(){
        console.log(a);
    }
    var a = 10;
    return inner;
}
outer()();
```

Yes, the code still forms a closure since sequence doesn't matter.

Q5. Changing var to let, will it make any differences?

```
function outer (){
    let a = 10;
    function inner (){
        console.log(a);
    }
    return inner;
}
outer()();
```

It will still behave the same way.

Q6. Will inner function have the access to outer function argument?

→
```
function outer (str){
    let a = 10;
    function inner (){
        console.log(a, str);
    }
    return inner;
}
outer("Hello world")();
```

→ Inner Function will now form closure and will have access to both a & str.

Q7) Output of below code and explanation?

```
function outest() {
  var c = 20;
  function outer(str) {
    let a = 10;
    function inner() {
      console.log(a, c, str);
    }
    return inner;
  }
  return outer;
}
let a = 100;
outest()("Hello")();
```

Output:

10 20 Hello

// Still the same output, the inner function will have reference to inner 'a', so conflicting name doesn't matter here. If it couldn't have find 'a' inside outer function then it could have went more outer to find 'a' and thus print 100.

Q8. Advantages of Closures:

→ Currying
→ Memoize
→ Datahiding & Encapsulation.
→ setTimeouts. etc.

Q9. Discuss more on Data Hiding & Encapsulation.

→ Data Hiding is like, suppose we have a variable and we want to have data privacy over it, so that ~~the outer~~ other functions or other pieces of code cannot access that variable.

E.g. // without closures

```
var count = 0;
function increment() {
    count++;
}
```

→ Anyone can access count and change it, no data-hiding involved.

// with closure → put everything into a function.

```
function counter() {
    var count = 0;
    function increment() {
        count++;
    }
}
console.log(count); → // This will result in reference error, as count can't be accessed. So, we are able to achieve data-hiding.
```

// Increment counter function using closure)
true function.

10/01/24

```
function counter(){
    var count = 0;
    return function increment(){
        count++;
    console.log(count); }

}
var counter1 = counter();
counter1();
var counter2 = counter();
counter2();
```

// Above code is not scalable for say, when you plan to ~~increment~~ implement decrement counter at a later stage.

→ we can use constructor to address this issue.

```
function Counter(){
    var count = 0;
    this.incrementCounter = function(){
        count++;
        console.log(count);
    }
this.decrementCounter = function(){
    count--;
    console.log(count); }
}
```

var counter1 = new Counter();
↳ new keyword for Constructor function.

counter1.increment Counter();
counter1.increment Counter();
counter1.decrement Counter();

// returns → 1 2 1

Q10. Disadvantage of closures?

→ Overconsumption of memory when using closure ~~as everytime~~ a everything And those closed-over variables are not garbage collected, till the program expires. If, not handled properly it could result in memory leaks, it can freeze the browser if not handled properly.

[Garbage collector]: Are programs in JS engine or Browser that frees up ~~this~~ unused memory.

Q11. Relation btw Garbage Collector & closure.

→
```
function a(){
    var x = 0;
    return function b(){
        console.log(x);
    };
}
```

```
var y = a();  // y is a copy of b.
y();
```

// Once a() is called, its element x should be garbage collected ideally. But function b() has closure over var x. So memory of x cannot be freed. Like this if more closure formed, it becomes an issue. To tackle this, JS engines like v8 and chrome have smart garbage collection mechanisms. Say we have var x = 0, z = 10 in above code. when console.log happens, x is printed as 0 but z is removed automatically.

13. First Class Functions ft. Anonymous Functions

Q1. what is Function Statement?

→ Below way of creating a function statement.

```
function a() {
    console.log("Hello");
}
a(); // Hello
```

Q2. What is Function Expression ?

→ Assigning a function to a variable. Function acts like a value.

```
var b = function () {
    console.log("Hello");
}
b();
```

Q3. Difference btw function statement and function expression.

→ The major difference btw these two lies in Hoisting.

```
a();  // Hello A
b();  // TypeError: b is not a function
function a() {
    console.log("Hello A");
}
var b = function () {
    console.log("Hello B");
};
```

Reason: During memory creation phase a() is created inside memory and whole function code assigned to a. But b is created as a variable (b: undefined) and until code reaches the function() part, it is still undefined. So, can't be called.

Q4. what is function Declaration?

→ Other name for function statement.

Q5. what is Anonymous function?

→ A function without name.

```
function () {
    ...
}
```

// this is going to throw Syntax error. Function statement requires function name.

→ Anonymous function are used when functions are used as values e.g the code sample for function expression above.

Q6. what is named function expression?

→ Same as Function Expression but function has a name instead of being anonymous.

```
var b = function xyz () {
    console.log("b called");
}
b(); // b called
xyz(); // Throws ReferenceError
```

xyz is not created in the global scope. So, it can't be called.

Q7. Parameters and Arguments:

```
var b = function (param1, param2) {
                    These are parameters
  console.log ("b called");
}
b(10, 20);
```

Arguments - values passed inside function call.

Q8. First Class Function

↳ Passing another function inside a function. Also we can return a function inside a function

```
var b = function (param1) {
  console.log (param1); // prints → "f() {}"
}
b (function () { });
```

// Another way:

```
var b = function (param1) {
  console.log (param1);
}
```

```
function xyz() {
}
b(xyz);
```

// We can return a function from a function.

```
var b = function (param1) {
    return function() { };
}
console.log(b()); // We log the entire function within b
```

In, Simple terms we can say, ~~***~~

- → Assigning functions to variables
- → Passing functions as arguments to other functions.
- → Return functions from other functions.

Functions are first class citizens means the same as first class functions.

14. Callback Function in JS.

ft: Event Listeners.

Functions are first class citizens, i.e taking a function 'A' and passing it to another function 'B'. Here, 'A' is a callback function. So, basically I am giving access to function B to call function A. This callback function gives us the access to whole Asynchronous world in Synchronous world.

```
setTimeout(function() {
console.log("Timer");
}, 5000); // first argument is callback
                function and 2nd is timer.
```

```
function x(y) {
  console.log("x");
  y(); }
x(function y() {
  console.log("y");
});
```

console:
x
y
Timer.

- In call stack, first x and y are present. After code execution, they go away and stack is empty. Then after 5 seconds (from beginning) anonymous suddenly appear up in stack, ~~it's called~~

~~blocking the main thread.~~
i.e setTimeout.

→ After 3 functions are executed through callstack. If any operation blocks the callstack, its called blocking the main thread.

→ Suppose, If a function xC), takes 30 seconds to run, the JS has to wait for it to finish as it has only 1 callstack / 1 main thread. Never block the main thread.

→ Always use async for functions that takes time eg. setTimeout.

Creating Event Listeners in JS.

→ We will create a button in HTML and attach event to it.

// index.html

```
<button id="clickMe"> Click Me! </button>
```

// index.js

```
document.getElementById("clickMe").addEventListener("click", function xyz(){
//when event click occurs, this callback function (xyz) is called into callstack.
console.log("Button Clicked");
});
```

Closure Demo with Event Listeners

Let's implement an increment counter button, we will use closure for data-abstraction.

```
function attachEventList(){
    let count = 0;
document.getElementById("clickMe")
.addEventListener("click", function xyz(){
    console.log("Button clicked", ++count);
// The callback function has formed
closure with outer scope (count).
});
}
attachEventList();
```

Garbage Collection and Remove EventListeners:

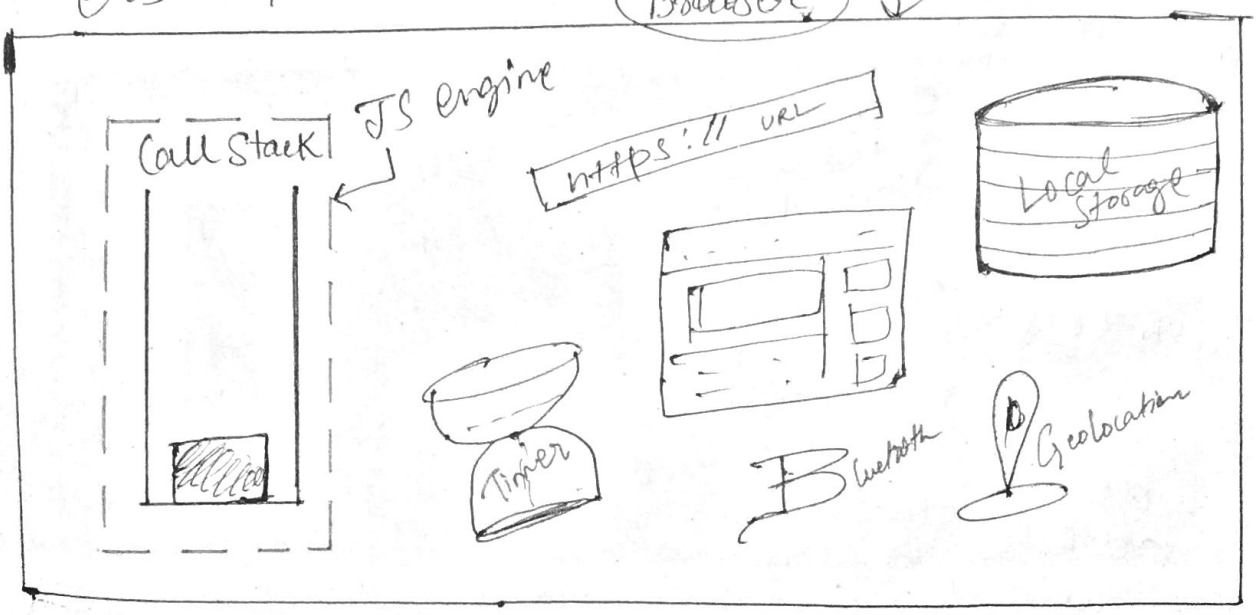
→ Event listeners are heavy as they form closures. So even when call stack is empty, EventListener won't free up memory allocated to count as it doesn't know when it may need count again. So we remove event listeners when we don't need them (garbage collected) onClick, onHover, onScroll all in a page can slow it down heavily.

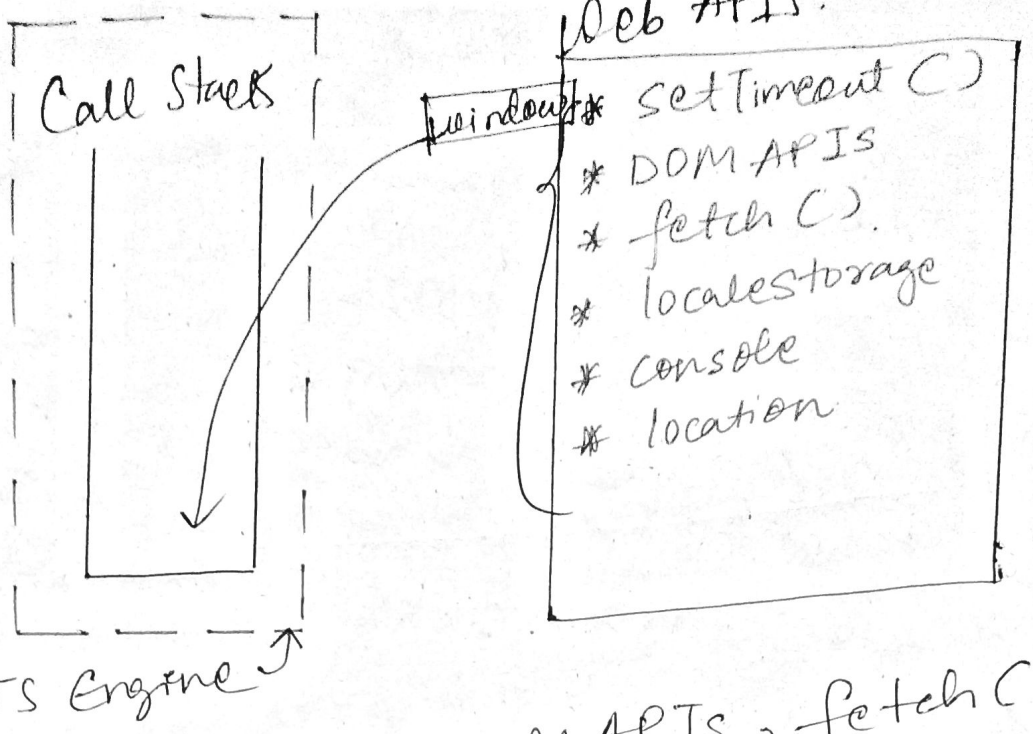
15. Asynchronous JavaScript & Event Loop

→ JS is a synchronous single-threaded Language.
→ It has one callstack.
→ It can do one thing at a time.
→ The call stack is present inside the JS engine
→ All code of JS is executed inside the call stack.

* Browser has JS Engine which has Call Stack, → global execution context inside call stack, local execution context etc.

→ But, browser has many superpowers - Local Storage space, Timer, place to enter URL, Bluetooth access, Geolocation access and so on.
→ Now JS needs some way to connect the callstack with all the superpowers. This is done using web APIs.





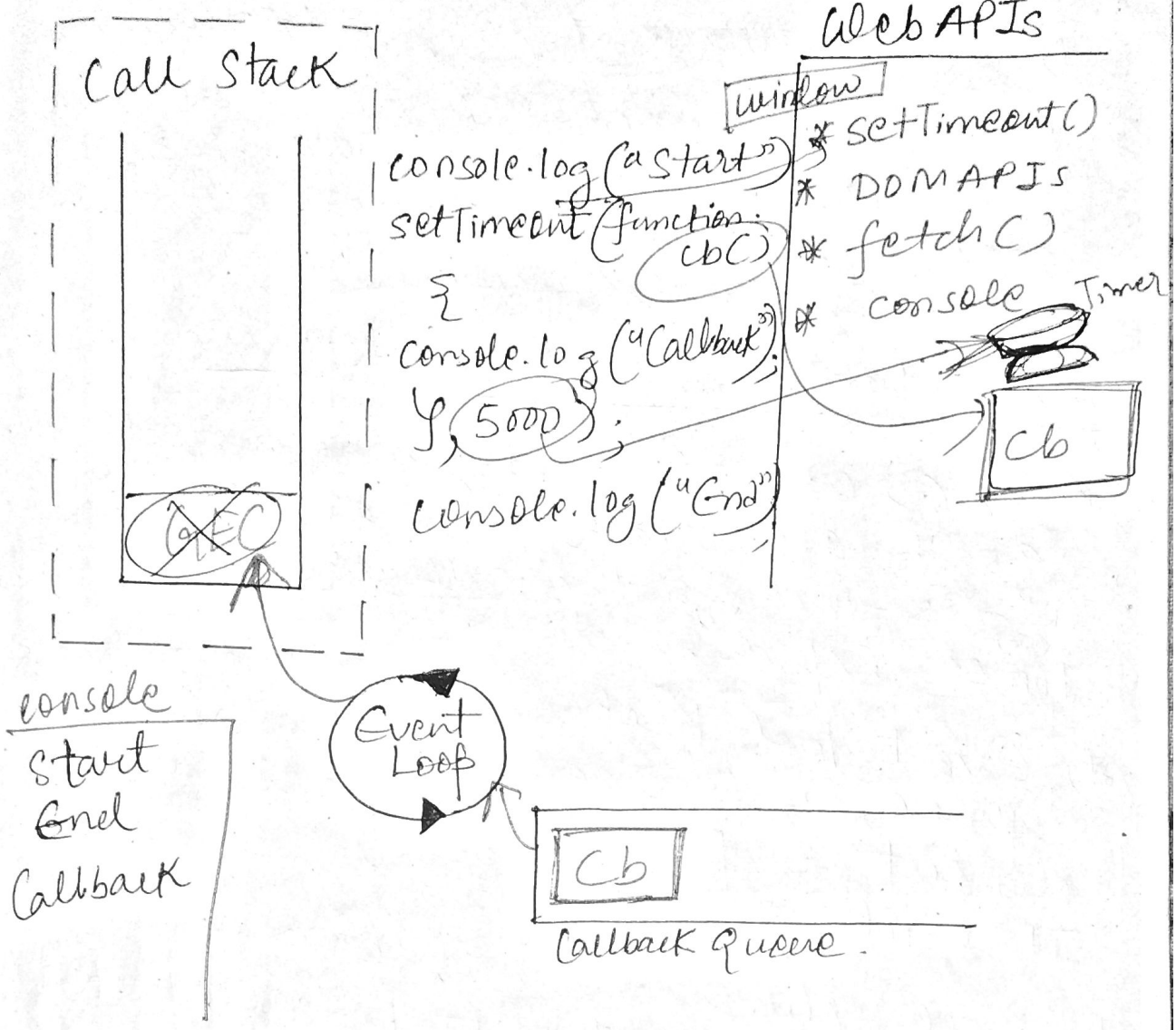
(*) setTimeout(), DOM APIs, fetch(), localestorage, console (yes, even console.log is not JS!!), location and so many more.

- setTimeout(): Timer Function
- DOM APIs: eg - document.getElementById("id name"), used to access HTML DOM tree. (Document Object Model).
- fetch(): used to make connection with external servers, eg. Netflix servers etc.

(*) We get all these inside call stack through global object i.e window

- use window keyword like: window.setTimeout(), window.localestorage, window.console.log() etc.
- As window is global object, and all the above functions are present in global object, we don't explicitly write window but it is implied.

How setTimeout works behind the scenes on Browsers ?



* First G.E.C is created and put inside the call stack.
* console.log("Start"); // this calls the console web api through window which in turn modifies values in console.
* setTimeout(function cbC() { // this calls the setTimeout web api which gives access to timer feature. It stores the cbC and starts timer. console.log("Callback"); }, 5000);
* console.log("End"); // calls console api and logs in console window. After this GEC pops from call stack.

* while all this is happening, the timer is constantly ticking. After it becomes 0, the callback cb() has to run.

* Now we need this cb to go into call stack. Only then will it be executed. For this we need eventLoop and Callback Queue.

Event Loops and Callback Queue.

* cb() cannot simply go to callstack to be executed. It goes through the callback queue when timer expires.

* Event loop keep checking the callback queue, and see if it has any element to put it into callstack. It is like a gate keeper.

* Once cb() is in callback queue, eventloop pushes it to callstack to run. Console API is used and log printed.

Another example: to understand EventLoop & Callback Queue.

How Event Listeners works in JS?

```
console.log("start");
document.getElementById("btn").
addEventListener("click", function cb(){
    console.log("Callback");
});
console.log("End");
```

→ cb() registered inside webapi environment and event (click) attached to it, i.e. REGISTERING CALLBACK AND ATTACHING EVENT TO IT.

~~console.log("callback");~~

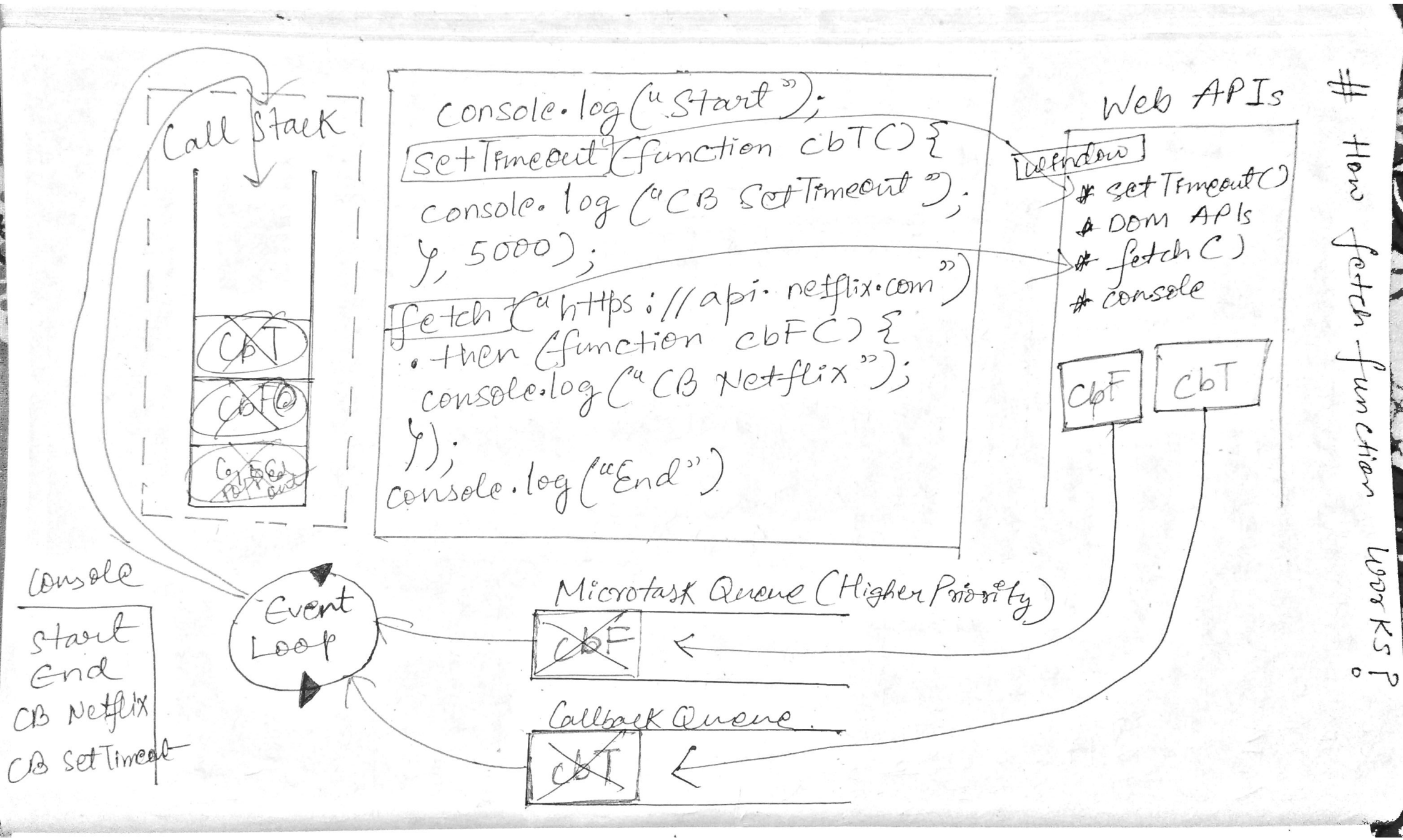
→ In the above code, even after console prints "start" and "end", and pops GEC out, the eventListener stays in webapi environment (with hope that user may click it some day) until explicitly removed, or the browser is closed.

Event Loop has just one main function:

It only job is to continuosly monitor the callstack as well as the callback queue, if the callstack is empty and the eventloop see that there is also a function waiting to be executed in the callback queue. It takes the function and pushes into callstack, and removes it from the queue.

Need of callback queue?

→ The callback queue in JavaScript is crucial for managing asynchronous operations and enabling non-blocking execution.


How fetch function work's ?
Call Stack
cbT
cbF
Console
console.log("Start");
setTimeout(function cbT(){
console.log("CB SetTimeout");
}, 5000);
fetch("https://api.netflix.com")
.then(function cbF(){
console.log("CB Netflix");
});
console.log("End")
Web APIs
window
setTimeout()
DOM APIs
fetch()
console
cbF
cbT
Event Loop
Microtask Queue (Higher Priority)
cbF
Callback Queue
cbT
Console
Start
End
CB Netflix
CB SetTimeout

Note : ***

→ Functions in Microtasks Queue are executed earlier than Callback Queue.

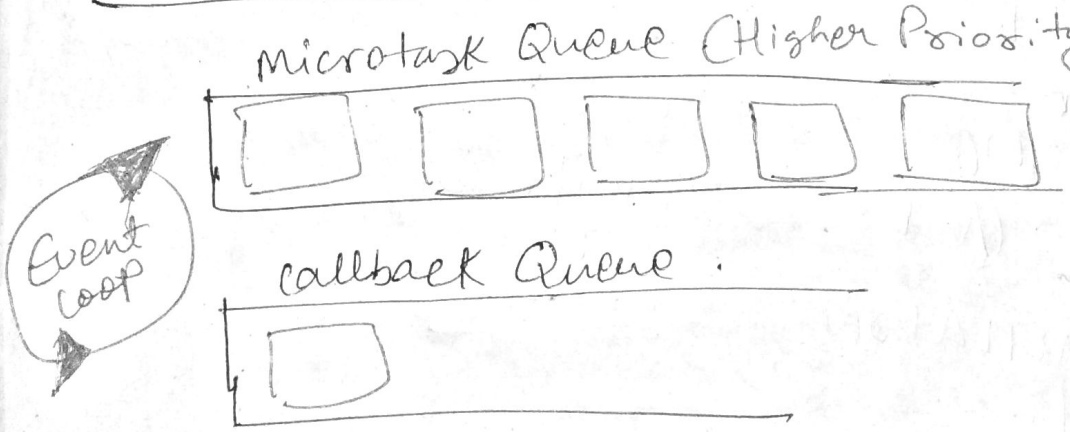
what enters Microtask Queue?

→ All the callback functions that come through promises go in microtask queue.

→ Mutation observer - It is a built in API that allows developers to asynchronously observe changes to the DOM. Keeps on checking whether there is mutation in DOM tree or not, and if there then it executes some callback function.

→ Callback functions that come through Promises and mutation observer go inside Microtask Queue, rest goes inside callback Queue.

Starvation of Callback Queue.



If, tasks in microtask queue keeps creating new tasks in the queue, element in callback queue never gets chance to be run. This is called Starvation.

16. JS Engine Exposed, Google's V8 architecture.

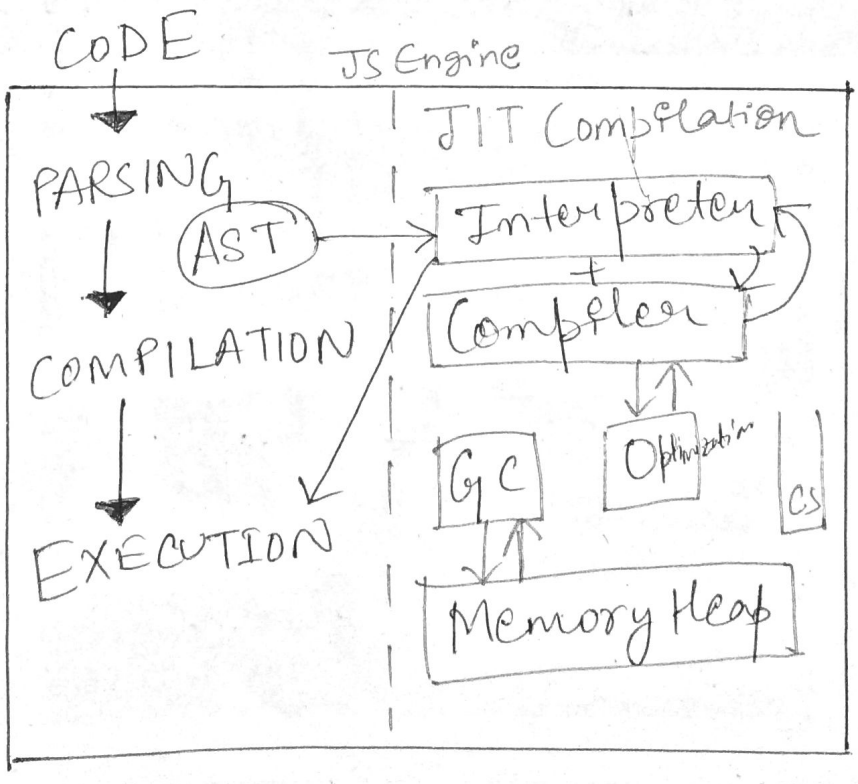
→ Javascript Runtime Environment is like a big container which has everything, which are required to run JS code.

→ JRE consists of JS Engine (Heart of JRE), set of APIs to connect with outside environment, eventloop, Callback queue, microtask queue etc.

→ Browser can execute JS code because it has the JRE.

JS Engine is not a machine. It is a software written in low level languages. Eg, Google's V8 engine (Fastest JS Engine.) is written in C++.

JS Engine architecture.

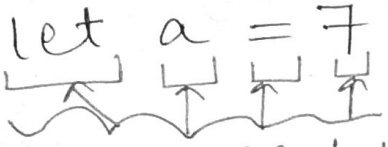


Code [redacted] inside JavaScript Engine
passes through 3 steps:

a) Parsing
b) Compilation
c) Execution

a) Parsing.

Code is broken down into tokens.



These are tokens.

Also, we have a syntax parser that takes code and converts into an AST (Abstract Syntax Tree), which is in a JSON format.

b) Compilation.

JS has something called Just-in-time (JIT) compilation — uses both interpreter & compiler. Also compilation and execution both go hand in hand. The AST from previous step goes to the interpreter, which converts high-level code to byte code and moves to execution. While interpreting, compiler also works hand in hand to compile and form optimized code during runtime.

c) Execution

Needs 2 components
- Memory Heap
- Call Stack

There is also a garbage collector. It uses an algo called Mark and Sweep.

17 Trust Issues with SetTimeout()

SetTimeout with Timer of 5 seconds sometimes does not exactly guarantees that the callback function will execute exactly after 5seconds.

Let's observe the code & explanation :-

```
console.log("Start");
setTimeout(function cb(){
  console.log("Callback");
}, 5000);
console.log("End");
// Millions of lines of code to execute.
.....
```

// Overhear setTimeout exactly doesn't guarantee that the callback function will be called exactly after 5s. Maybe 6, 7 or even 10! It all depends on the callstack. Only!

Reason:

→ First GEC is created and pushed in callstack.

→ Start is printed in console.

→ When setTimeout is seen, callback function is registered into web api's environment. And timer is attached to it and started. Callback waits for its turn to be executed once timer expires. But JS waits for none. Goes to next line.

→ End is printed on console.
→ After "End", we have 1 million lines of code that takes 10sec (say) to finish execution. So GEC won't pop out of the call stack. It runs all the code for 10 sec.
→ But in the background, the timer runs for 5 seconds. while callstack runs the 1M line of code, this timer has already expired and callback function has been pushed to ~~callstack~~ callback Queue and waiting to pushed to callstack to get executed.
→ Event Loop keeps checking if callstack is empty or not. But here GEC is still in stack so [cb] can't be popped from callback queue and pushed to callstack. Though setTimeout() is only for 5s, it waits for 10s until callstack is empty before it can execute.
→ This is called as Concurrency model of JS. This is the logic behind, setTimeout's trust issues.

18. Higher-Order Functions.ft.

Functional Programming.

Q. What is Higher Order Function?

→ A higher order functions are regular functions that take other functions as arguments or return functions as their results. eg:-

```
function x() {
    console.log("Hi");
};
function y(x) {
    x(); }
};
y();
y(); // Hi.
// y is a Higher order function.
// x is a callback function.
```

Let's try to understand how we should approach solution in interview I, have an array of radius and I have to calculate area using these radius and store in an array.

Naive Approach

```
const radius = [1, 2, 3, 4, 5];
const calculateArea = function (radius)
{
  const output = [];
for(let i = 0; i < radius.length; i++)
  {
  output.push (Math.PI * radius[i] *
                          radius[i]);

  }
  return output;
};
console.log (calculateArea (radius));
```

The above code solution works perfectly fine but what if we have new requirement to calculate array of circumference. Code now will be -

```
const radius = [1, 2, 3, 4, 5];
const calculateCircumference = function (radius)
{  const output = [];
for(let i = 0; i < radius.length; i++) {
  output.push (2 * Math.PI * radius[i]);
  }
  return output;
};
console.log (calculateCircumference (radius));
```

// Here, the both 2 codes for area and circumference are absolutely correct, but there is a lot of repetitiveness of code. We can optimize it using Functional Programming.

```
const radiusArr = [1, 2, 3, 4, 5];

// logic to calculate area.
const area = function (radius)
{
return Math.PI * radius * radius;
}

// logic to calculate circumference.
const circumference = function (radius)
{
return 2 * Math.PI * radius;
}

const calculate = function (radiusArr, operation)
{
  const output = [];
for (let i = 0; i < radiusArr.length; i++)
{
output.push (operation (radiusArr[i]));
}
return output; }
```

```
console.log(calculate(radius, area));
console.log(calculate(radius, circumference));
```

Higher Order function

Polyfill of Map

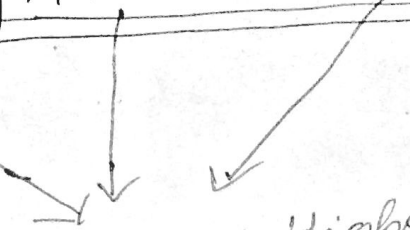
// Over here, calculate is nothing but polyfill of map function,

```
// console.log(radius.map(area));
console.log(calculate(radius, area));
```

Let's try to convert above calculate function and try to use. So,

```
Array.prototype.calculate = function (operation) {
  const output = [];
  for (let i = 0; i < this.length; i++) {
    output.push(operation(this[i]));
  }
  return output;
}
console.log(radius.calculate(area));
```

19. map, filter & Reduce.



These are Higher order Function.

map function

It is basically used to transform an array. The map() method creates a new array with the results of calling a functi for every array element.

Example:

```
const arr[] = [5, 1, 3, 2, 6]

// Task 1: Double the array elements
        [10, 2, 6, 4, 12]

function double (x) {
    return x*2; }

const doubleArr = arr.map(double);
```

Internally map will run double function for each element of array and create a new array and returns it.

```
console.log(doubleArr); // [10, 2, 6, 4, 12]
```

// Task 2: Triple array elements.

```
function triple(x){
  return 3*x;}

const tripleArr = arr.map(triple);
console.log(tripleArr); // [15, 3, 9, 6, 18]
```

// Task 3: Convert array element to Binary

or

```
e. function binary(x){
  return x.toString(2);}

const binaryArr = arr.map(binary);
```

Way 1

The above code can be written as :-

```
const binaryArr = arr.map(function binary(x){
    return x.toString(2); });
```

Way 2.

```
const binaryArr = arr.map((x) => x.toString(2));
```

Way 3.

filter function - It is basically used to filter the value inside an array.

```
const array = [5, 1, 3, 2, 6];
// Filter odd values.
function isOdd(x) {
    return x % 2; }
const oddArr = array.filter(isOdd);
```

↳ way 1

```
const oddArr = array.filter((x) => x % 2);
```

filter function creates an array and store only those values which evaluates to true.

Reduce function - It is a function which take all the values of array and gives a single output of it. It reduces the array to give a single output.

Examples:

```
const array = [5, 1, 3, 2, 6];
// Calculate sum of elements of array → Non-Functional
   Programming Way.
function findSum(arr) {
    let sum = 0;
for(let i = 0; i < arr.length; i++) {
    sum += arr[i]; }
```

return sum; }
console.log(findSum(array)); // 17

// Using reduce function.

const sumOfElem = array.reduce(function
(accumulator, current) {

// current represents the value of array.
// accumulator is used to get result from array elements.

// In, comparision to previous code snippet, 'sum' variable is 'accumulator' here, and 'arr[i]' is 'current'.

accumulator = accumulator + current;
return accumulator;

}, 0);
↑ accumulator's initial value

console.log(sumOfElem);

Example 2:

// Find max element in an array

way 1: Non-Functional Programming.

const array = [5, 1, 3, 2, 6];

function findMax(arr) {
let max = 0;
for (let i = 0; i < arr.length; i++) {
if (arr[i] > max) {
max = arr[i]; } }

```
return max; }
console.log(findMax(array)); // 6

way 2: Using reduce function.

const maxEle = array.reduce((accumulator, current) =>
    {
        if(current > accumulator) {
        accumulator = current; }
        return accumulator;
    }, 0);
console.log(maxEle); // 6.

// Trick MAP Example:
  const users = [
{firstName: "Sayan", lastName: "Raha", age: 23},
{firstName: "Sumit", lastName: "Verma", age: 20},
{firstName: "Rahul", lastName: "Raul", age: 40},
{firstName: "Arun", lastName: "Pandey", age: 22},
];
// Get array of fullNames.
const fullNameArr = users.map((x) =>
                        x.firstName + " " + x.lastName);
console.log(fullNameArr);
```

// TRICKY REDUCE EXAMPLE.

// Get the count/report of how many unique people with unique age.

[22: 2, 20: 1, 40: 1]

```
const report = users.reduce(
    (accumulator, current) => {
        if (accumulator[current.age]) {
            accumulator[current.age] = ++accumulator[current.age];
        }
        else {
            accumulator[current.age] = 1;
        } return accumulator;
    }, {});
console.log(report);
```

// Function Chaining

↳ Find firstName of all people whose age is less than 30. — using same array.

```
const users = [
{firstName: "Alok", lastName: "Raj", age: 23},
const output = users.filter((user) =>
    user.age<30).map((user) =>
    user.firstName);
console.log(output);
// ["Sayan", "Samit", "Arun"]
```

// Homework: Implement the same logic usi...

Reduce

```
const output = users.reduce((acc, curr) =>
    if (curr.age < 30) {
        acc.push(curr.firstName);
    }
    return acc;
}, []);
console.log(output);
```